Distributed synchronization

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

Understanding system state – 1

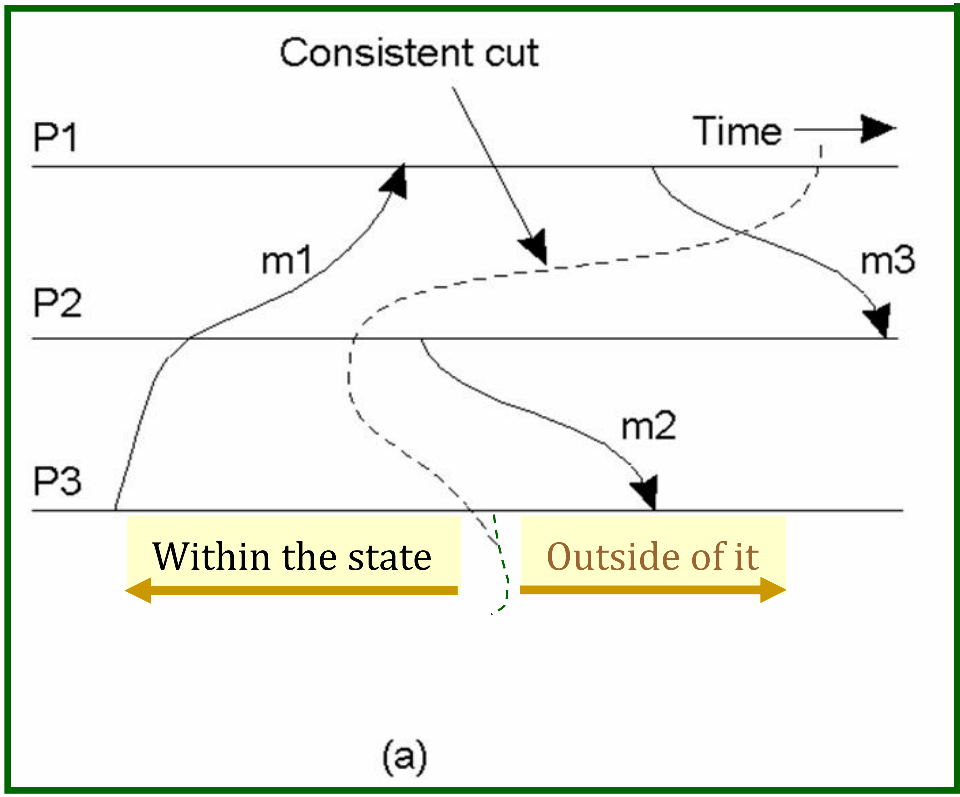
- The *global* state of a distributed system is comprised of two distinct parts
  - Selected elements of *local* states
  - All inter-node messages currently *in flight*
- Knowing the global state helps coordinator agents
  - To detect the presence vs absence of activity
    - No in-flight messages suggest lack of global activity
  - To diagnose the causes of absence of activity
    - Normal termination vs abnormal stall

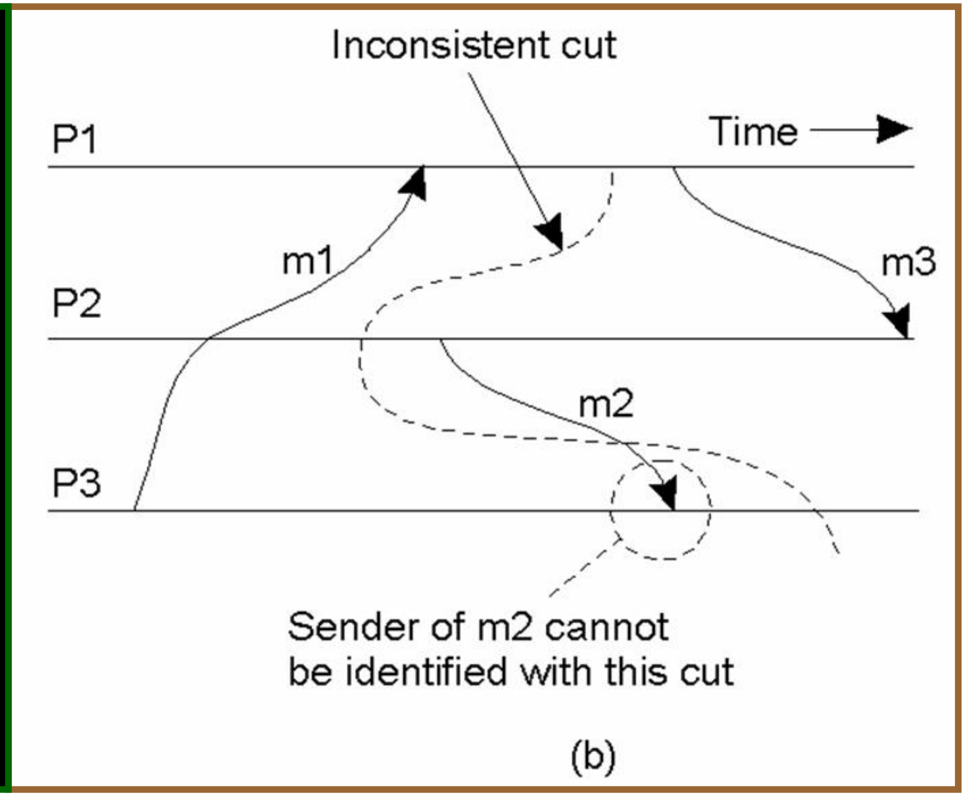
Understanding system state – 2

- Global state is captured with a **distributed snapshot** (pun intended ☺) that provides a **consistent** representation of *a* "true" global state
  - Capable of *causing* progress that conforms with system spec
- It is a **causal** notion, *not* an instantaneous-time concept
  - Which is cannot be without shared memory !
  - A local state in node B that includes the reception of a message not sent in the sender's local state in node A is **not consistent**
- It is realised as a "cut" in the temporal succession of all **individual** local states
  - It tells what falls in the global state and what does not
  - It does *not* require the use of a global-time line
    - Because in general it cannot assume there can be one …

Understanding system state – 3

K. Chandy, L. Lamport
Distributed Snapshots: Determining Global States of Distributed Systems
ACM Transactions on Computer Systems, 3(1):63-75, 1985





Understanding system state – 4

- Building a consistent cut requires telling apart
- **Inconsistent messages**
  - Sent by node S **after** the latest local checkpoint, but received by node R **before** the latest local checkpoint
    - A distortion of causality consequent to lack of instantaneity
    - “Relevant“ local checkpoint belongs in the distributed snapshot of interest
  - Restoring the system from that inconsistent-cut state would cause S to re-send that message, outside of specification
    - Harmful unless R’s action on reception was idempotent …
- **In-flight messages**
  - Those sent by S **before** the latest local checkpoint, whose arrival is not recorded in the latest local checkpoint at R

- A distributed snapshot contains **no** inconsistent messages

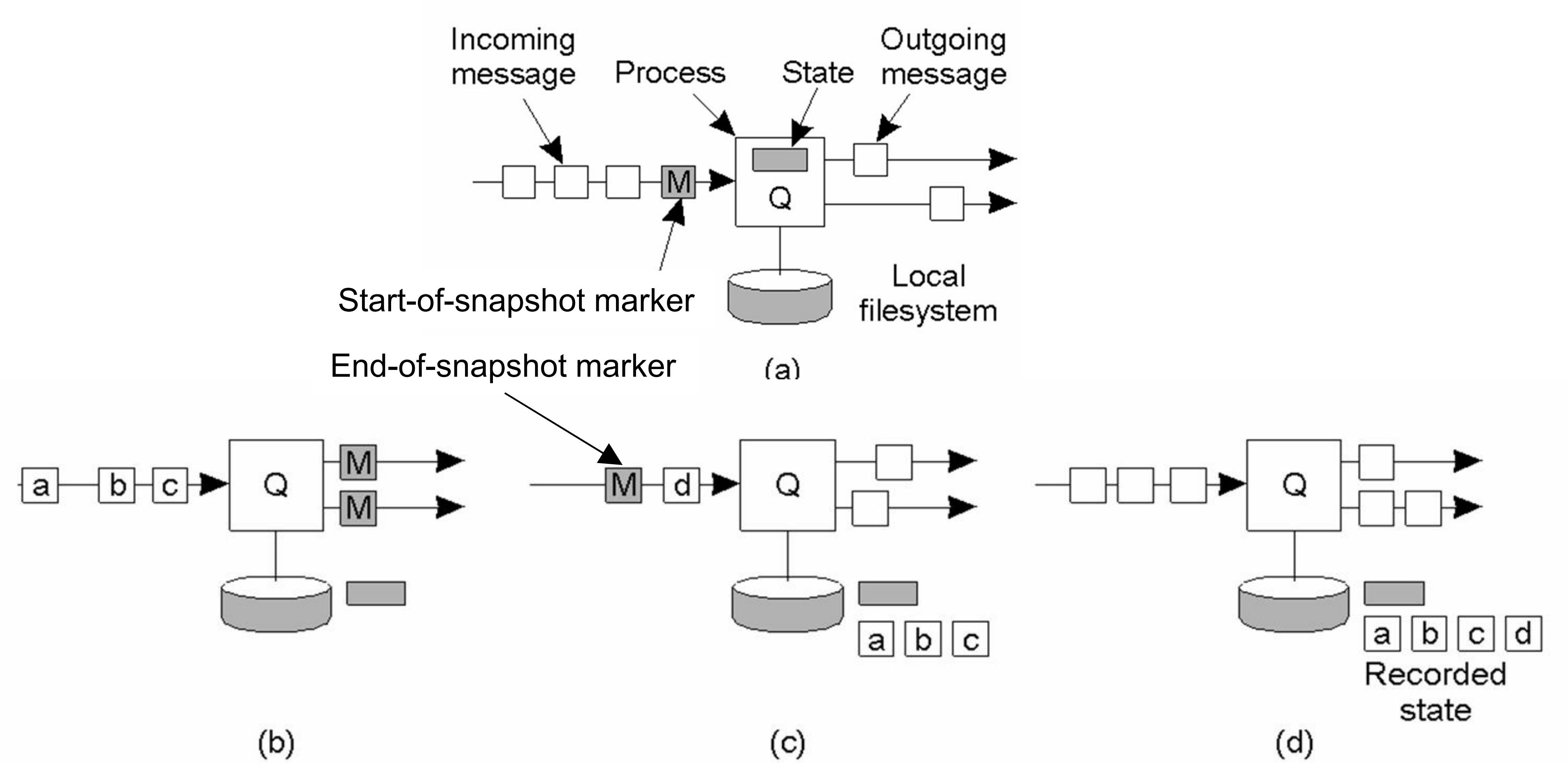
Taking a distributed snapshot – 1

- System is comprised of nodes connected by point-to-point channels in an *overlay* network
  - All nodes reachable in a finite number of hops
  - Every node is a multi-threaded process
- Any node may initiate a distributed snapshot
  - No coordination required
    - Snapshots are permissionless and may run in parallel
  - Initiator saves local state and sends a marker down all of its outbound channels
    - The marker identifies initiator and current snapshot

Taking a distributed snapshot – 2

- A node that receives a start-of-snapshot marker
  - Stores local state (if not saved already), *suspends local work*, forwards marker down all of its outbound channels
    - Remember: multiple snapshots may run in parallel
  - Saves locally all in-flight messages that are hopping to their destination
    - Not forwarding them helps create quiescence
  - Until it receives relevant end-of-snapshot marker
    - Which it forwards to its successor nodes
    - And posts its complete local state onto a designated global place, with "finished" notification to initiator

Taking a distributed snapshot – 3



Why does this algorithm always produce a consistent cut?

Use case: synchronized termination

- The system topology yields $n \geq 1$ directed reachable graphs rooted in all "initiator" nodes
  - Minimum baseline is a single directed graph for one initiator
- Node Q that receives "start-termination" marker μ from node M
  - Forwards μ down all of its outbound channels
  - Makes its own logical local shutdown
  - Awaits "finished" messages from all of its successor nodes
  - Sends M a "finished" message when that happens **as long as** Q has not seen further in-flight messages meanwhile
    - Otherwise Q sends M a "continued" message and M may retry
- The global effect of M's quest occurs when receiving "finished" messages from *all* of its successor nodes

Demo implementation in Ada

- Whole system simulated as a single concurrent program
- Each node is pair of nested tasks
  - Parent handles inbound messages
  - Child sends work messages down outbound channels
- Communication channels are unidirectional
  - Implemented with entries
- Topology is a directed *reachable* graph rooted in node 1
- At a given point, node 1 sends termination marker out
  - Nodes that receive marker start recording their local state and then instigate local termination
- Graceful termination happens across nodes as
  - Node's child task ends when local state has been recorded
  - Node's parent task "accepts termination" when no sender is left

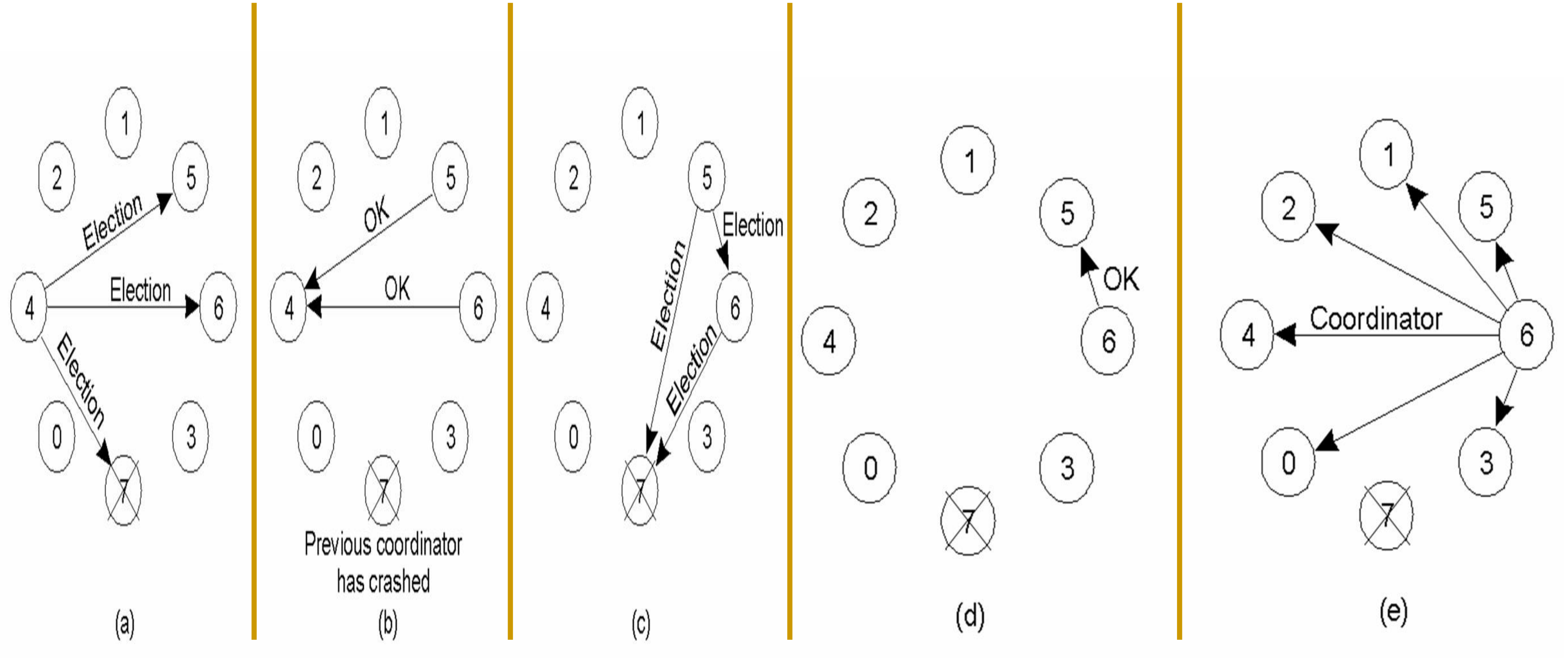
Leader election

- Having a leader simplifies distributed algorithms
  - But leader must be there when needed
  - It needs to be elected anew if lost or unreachable …
- Leader election requires distributed consensus
  - Election algorithm must assure termination with majoritarian agreement
- Prerequisites
  - A unique fully-ordered ID per node
  - Every node knows the ID of all other nodes
    - Dynamically arriving or leaving participants complicate the problem a lot

Leader election: the bully algorithm – 1

- A node P that does not know the leader calls an election
  - Resuming after halt or skipping leader's heartbeats
- P sends an "Election" message to all nodes with **higher** ID
  - On receiving "Election" message from sender with **lower** ID, node Q responds "My job", and initiates new election
  - On receiving "My job" reply, sender quiets itself
- If **no** node replies, the sender becomes the leader
  - The new leader node begins to notify the other nodes
- Leader is **always** the node with ID greater than all currently alive and reachable nodes

Leader election: the bully algorithm – 2



The consensus problem – 1

- Partitioning responsibilities and data helps scale on the Y and Z axes of the scalability cube
  - **Exam theme #1**: explore scalability challenges and solutions in a chosen distributed application or service
- But also makes assuring state consistency a much harder problem …
  - Erroneous views may start circulating: how can they be prevented and rectified
- Solutions are needed that assure consistency of system status (and output)
  - Singling out one value strictly among those that participants actually proposed
    - No self-generated proposals, no pretended notifications

The consensus problem – 2

- A most famous and influential solution to this problem, nicked "**Paxos**", can be traced to
  - L. Lamport, *The part-time parliament*, ACM TOCS 16(2), 1998, doi: 10.1145/279227.279229
  - **Exam theme #2**: apply Paxos or its variant **Raft**[1] to a real-world PoC problem of your choice

[1]: https://raft.github.io/

Distributed access control – 1

- **Centralized** solution: easy but fragile
  - A leader is assumed, which receives all access requests for any shared resource anywhere
  - Node P requesting access to resource R sends "**May I?**" message to leader
  - If resource is free, leader responds "**Granted**"
    - Else it responds "**Denied**" and stores request in FIFO queue
    - Receiver node holds
  - On relinquishing R, node Q sends "**Released**" message to leader
    - Leader sends "Granted" to node whose request is head of queue
- Coordinator is **single point of failure** and bottleneck

Distributed access control – 2

- **Distributed** solution
  - Node P seeking access to resource R sends message $\mu_P = \langle \rho, P, R, c_P \rangle$ to **all** other nodes, with c_P timestamp at P
  - Node Q receiving μ
    - If not interested in R, replies “OK”
    - If holding R, it does **not** reply, adding μ_P to local wait queue for R
      - On relinquishing R, it sends “OK” to all nodes with requests in queue
    - If it requested access to R with $\mu_Q = \langle \rho, Q, R, c_Q \rangle$ **without** being granted it yet, it checks c_P against c_Q
      - It replies “OK” if $C_P \leq C_Q$
  - Node P grabs R only after receiving “OK” from **all** other nodes
- Every node is one **single point of failure**
  - Protocol traffic increases considerably
- Decision on timestamps requires some degree of ordering
  - L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, CACM 21(7), 1978, doi: 10.1145/359545.359563

Distributed access control – 3

- Another distributed solution
  - Nodes are ordered in a **ring** topology
    - A **circulating token** grants exclusive access to **single** shared resource
  - Node 0 generates token and starts circulating it
    - Node receiving token may grab resource, then it must pass token along to successor on ring
    - Node receiving token acknowledges to predecessor
      - Ring bypasses node that fails to acknowledge
  - Worst-case wait time is one full round of the ring
- Token is **single point of failure**
  - Lost token must be generated anew
    - When a node does not “see” it within bounded time

Distributed access control: comparison

Variant	# Messages between request and release	Worst-case overhead for message sending	SPoFs
Centralized	**3** (ENTER, GRANTED, RELEASED)	**2** (ENTER, GRANTED)	**Coordinator**
Distributed	**2 (n – 1)** (GRANT?, RELEASED)	**2 (n – 1)**	**Any node**
Token ring	**1 .. ∞** (worst case when no node wants access)	**0 .. n – 1** (worst case when token must make a full round)	**Token**

Excluded topic of particular interest

- **Distributed transactions** (two-phase commit) are costly
  - They may cause heavy bottlenecks and massive decrease of throughout, hence scarce availability
  - Their model is known as **strict consistency**
  - **Exam theme #4**: explore the rationale of the **saga pattern** for microservice architectures, and the challenges of adopting it for real
  - Interesting initial reads
    - https://microservices.io/patterns/data/saga.html
    - https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga
- **Eventual consistency** is a an attractive alternative
  - Much better availability, when users can afford uncertainty
  - The paradigm of choice for **NoSQL** databases
  - **Exam theme #3**: study where and how eventual consistency is used, and make critique of it
  - Interesting initial reads
    - https://www.oracle.com/technetwork/consistency-explained-1659908.pdf
    - https://medium.com/swlh/handling-eventual-consistency-11324324aec4